Data type qualifier

Data types in C can come with *modifiers* or *qualifiers*. There are four of these, namely *short*, *long*, *signed* and *unsigned*. The physical quantities associated with these qualifiers differ from one implementation to another. But a `short int` may either have word of a length shorter than that of an *int*, or one equal to the same. The qualifier *signed* means that the leftmost bit in each word is reserved for the sign. Thus the maximum absolute value of a *long int* is approximately twice that of an *int*, though the range or the span of both remains the same. Table 1 gives the combinations of variable types and modifiers. In particular *short int*, *long int* and *unsigned int* are often called simply *short*, *long* and respectively *unsigned*. The *bytes* and *range* fields represent the typical values of these.

Description	*Signedness*	*Bytes*	*Range*	*Also known as*
`char`	signed	1	-128–127	`signed char`
`int`	signed	2	-32768–32768	`signed int`
`short int`	signed	2	-32768–32768	`short`
`long int`	signed	4	-2147483648–2147483647	`long`
`unsigned char`	unsigned	1	0–255	
`unsigned int`	unsigned	2	0–65535	`unsigned`
`unsigned short`	unsigned	2	0–65535	`unsigned short int`
`unsigned long`	unsigned	4	0–4294967295	`unsigned long int`
`signed long`	signed	4	-2147483648–2147483647	`signed long int`
`enum`	unsigned	2	0–65535	
`float`	signed	4	$3.4e \pm 38$ (7 digits)	
`double`	signed	8	$1.7e \pm 308$ (15 digits)	
`long double`	signed	10	$3.4e - 4932$–$1.1e4932$	

Table 1 *Variable types with modifiers*

Compiler and compilation

A computer programme in general does three things, that is input, process and output. A compiler itself is also a programme. Its task is to produce another programme, its input being the source code for that programme. There are four stages in the compilation process, namely *preprocessing*, *compiling*, *assembling* and *linking*. Each of these stages is in turn a programme in its own right, having its input, processing and output. The link editor combines the otherwise executable object code with function calls kept in some other sources. Figure 1 shows the flow chart of stages in compilation.

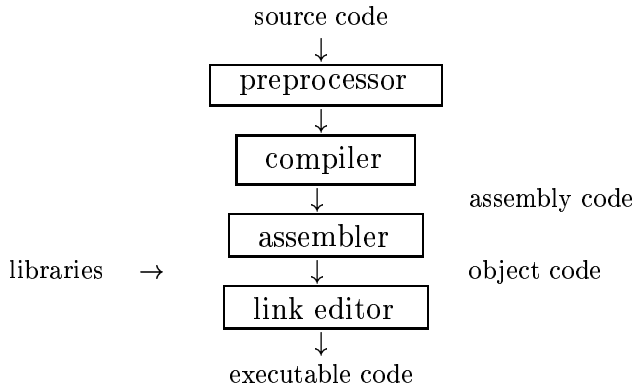

Figure 1 *Compilation stages*

The preprocessor has components and their interaction as described in Figure 2. The *lexical analyser* is also called a *scanner*, the *syntax analyser* a *parser*. *Tokens* are such entities as *keywords*, *variable names*, *constants* and *operators*. Examples of keywords are `while`, `do` and `for`. The *syntax analyser* determines the structure of the programme according to a grammar. A *syntax tree* has tokens as its leaves, and every one of its nonleaf nodes a syntactic class type. As an example, the intermediate source code of the *infix* expression

$$(a + b) * (c + d)$$

could be the following *prefix* expression in sets of quadruples,

$$(+, a, b, t_1)(+, c, d, t_2)(*, t_1, t_2, t_3)$$

or the suffix expression *Polish* notation

$$ab + cd + *$$

Here t_1, t_2 and t_3 are temporary variables.

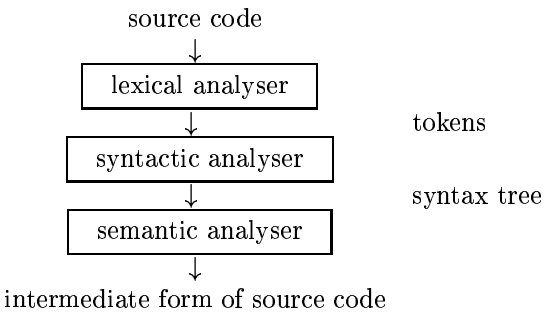

Figure 2 *Components of a preprocessor*

Operator precedence

The various operators have different levels of precedence or priority. For example, the expression $a * b + c * d$ would be interpreted as $(a * b) + (c * d)$. Table 2 shows the different operators together with their relative priority.

Operator	*Description*	*Associativity*
`()`	function call	left to right
`[]`	array element	left to right
`.`	structure member	left to right
`->`	pointer to a structure member	left to right
`!`	logical NOT	right to left
`~`	one's complement	right to left
`-`	minus	right to left
`++`	increment	right to left
`--`	decrement	right to left
`&`	address of	right to left
`*`	contents of	right to left
`(variable type)`	type cast operator	right to left
`sizeof`	returns size in bytes	right to left

Table 2 *Operator precedence*

Operator	*Description*	*Associativity*
`*`	multiply	left to right
`/`	divide	left to right
`%`	modulus	left to right
`+`	add	left to right
`-`	subtract	left to right
`<<`	leftward shift	left to right
`>>`	rightward shift	left to right
`<`	less than	left to right
`<=`	less than or equal to	left to right
`>`	greater than	left to right
`>=`	greater than or equal to	left to right
`==`	equal to	left to right
`!=`	not equal to	left to right
`&`	bitwise AND	left to right
`^`	bitwise XOR	left to right
`|`	bitwise OR	left to right
`&&`	logical AND	left to right
`||`	logical OR	left to right
`?:`	conditional	right to left
`=`	assignment	right to left
`*=, /=, %=, +=`	compound assignment	right to left
`-=, <<=, >>=`	compound assignment	right to left
`&=, ^=, !=`	compound assignment	right to left
`,`	comma	left to right

Table 2 *Operator precedence (continued)*

Array

An *array* is a group of data organised into an orderly grid referenced with integral indices. Example 1 shows addresses of members of a one-dimensional array whose members are integers. The outputs of this programme are given in Figure 3.

Elements of an array in one dimension are shown in Example 1 in the form $a[i]$. Similarly if our array has instead three dimensions we could perhaps write it in a form $a[i][j][k]$, where i, j and k are integer indices.

The output shown in Figure 3 says that the size of an integer is 4. This is the value the GCC Version 3.3.5 implementation uses. Compare this with the value 2 of `sizeof(int)` given in Table 1.

Example 1. (Members' addresses of an array)

```
1 #include<stdio.h>
2 int main(){
3   int i, a[10];
4   printf("\n size of int is %d\n\n", sizeof(int));
5   for(i=0; i<=9; i++){
6     printf("&a[%d] = %x\n", i, &a[i]);
7   }
8   return 0;
9 }
```

```
kit@nebula:~/prog/c$ tst
 size of int is 4
&array[0] = bffffab0
&array[1] = bffffab4
&array[2] = bffffab8
&array[3] = bffffabc
&array[4] = bffffac0
&array[5] = bffffac4
&array[6] = bffffac8
&array[7] = bffffacc
&array[8] = bffffad0
&array[9] = bffffad4
```

Figure 3 *Outputs of Example 1*

Exercise

Exercise 1. Let p, q and r be integers, and let $p = 7$, $q = 5$ and $r = -3$. Find the values of the following expressions.

(a)	$a + b - c$	(b)	$a\%b$	(c)	a/b
(d)	a/c	(e)	$a * (b/c)$	(f)	$a * b/c$
(g)	$a\%(b * c)$	(h)	$a\%b * c$	(i)	$x\%y + 1\%z$

Exercise 2. Let x, y and z be floating-point variables, and let $x = 10.1$, 6.5 and -2.3.n Find the values of the following expressions.

(a)	$(x/y) + z$	(b)	$x/y + z$	(c)	$x/(y + z)$
(d)	$3 * x/5 * z$	(e)	$x\%y * z$	(f)	x*x%y
(g)	$x\%y$	(h)	$x * (y - (z/2))$	(i)	$x/y/z * 2 - z * x$

Exercise 3. Let c, d and e be variables of character type, and let $c =$ 'F', $d =$ '11' and $e =$ '$'. Find the numerical values of the following expressions.

(a)	$c * d + e$	(b)	$d +$ '#'	(c)	$c\%d$
(d)	$('7' * d/c) + e$	(e)	$c * d/e$	(f)	$c + d + e -$ '1'
(g)	'1' + '1'	(h)	'a' + 'a'	(i)	$c *$ 'c' $+ d/$'d'

Exercise 4. Do Exercises 1, 2 and 3 again using values of variable inputted from keyboard.

Problem 1. Looking at how `sizeof` is used in the programme used in Example 1, find the actual sizes implemented in our compiler of the various variable types.

Bibliography

Byron Gottfried. *Programming with C.* Schaum's Outlines Series, McGraw-Hill, 1996

Steven Holzner. *C Programming.* Brady, 1991

Robert C Hutchison and Steven B Just. *Programming using the C language.* Computer Science Series, McGraw-Hill,

Jean-Paul Tremblay and Paul G Sorenson. *The theory and practice of compiler writing.* McGraw-Hill, 1985